5 CONCLUSIONS AND FUTURE WORK

You're bound to be unhappy if you optimize everything.

— Donald Knuth

Owing to the growing adoption of WebAssembly, we have focused on the security of WebAssembly programs and the potential usages of Software Diversification. This thesis introduces a comprehensive set of methods and tools for Software Diversification in WebAssembly. It includes the technical contributions of this dissertation: CROW, MEWE, and WASM-MUTATE. Additionally, we present specific use cases for exploiting the diversification created for WebAssembly programs. In this chapter, we initially summarize the technical contributions of this dissertation, including an overview of the empirical findings of our research. Finally, we discuss future research directions in WebAssembly Software Diversification.

5.1 Summary of technical contributions

Our first tool, CROW, is a compiler-based approach. It uses the LLVM compiler and requires the source code or the LLVM IR representation for its functioning. Its core comprises an enumerative synthesis implementation. CROW ensures the functional equivalence of the generated variants by employing SMT solvers for functionality verification.

MEWE, on the other hand, enhances CROW by using identical core technology to generate program variants. Furthermore, it encapsulates the LLVM IR variants into a WebAssembly multivariant binary, facilitating execution path randomization. Both CROW and MEWE are fully automated systems, necessitating only the input source code from users.

WASM-MUTATE, a binary-based approach, uses a set of rewriting rules and the input Wasm binary to generate program variants. In WASM-MUTATE, the generation of WebAssembly variants primarily involves random e-graph

[0] Compilation probe time 2023/10/31 10:25:49

traversals. Remarkably, WASM-MUTATE eliminates the need for compiler adjustments, thus ensuring compatibility with all existing WebAssembly binaries. Unlike CROW and MEWE, which are confined to code and function sections, WASM-MUTATE can generate variants by transforming any segment of the Wasm binary.

CROW, MEWE, and WASM-MUTATE are open-source, public tools, making their deployment entirely practical. Notably, WASM-MUTATE is currently in use in real-world scenarios to enhance WebAssembly compilers[1].

5.2 Summary of empirical findings

According to the comparison of our technical contributions discussed in Chapter 3 and, the results of our use case experiments in Chapter 4 we summarize the following empirical findings.

Implications of our implementations: CROW and MEWE depend on SMT solvers to prove functional equivalence in their enumerative synthesis implementation, which can be a bottleneck in variant generation. Consequently, WASM-MUTATE outperforms CROW and MEWE by producing unique variants. It achieves this in at least an order of magnitude greater, within the same timeframe. The main reason is that WASM-MUTATE uses a preset of rewriting rules accompanied by virtually inexpensive random e-graph traversals. The applications of our technical contributions are not orthogonal but complementary. Specifically, one can employ CROW and MEWE to generate a set of variants, which subsequently serve as rewriting rules for WASM-MUTATE. Furthermore, when practitioners require swift generation of variants, they could utilize WASM-MUTATE, accepting a decrease in preservation of the variants.

Offensive Software Diversification: We use WASM-MUTATE to illustrate the practical application of Offensive Software Diversification in WebAssembly. Specifically, we employ WASM-MUTATE in generating WebAssembly variants of cryptojacking malware. These variants effectively evade detection from state-of-the-art malware detection systems like VirusTotal and MINOS. Our research verifies the existence of opportunities for the malware detection community to bolster the automatic detection of cryptojacking WebAssembly malware. One potential contributing factor to the success of WASM-MUTATE's evasion is a false sense of resilience. Prior research into the detection of WebAssembly malware has exposed a flawed presumption that obfuscation techniques for WebAssembly are absent [**? ? ? ? ?**], whilst our software diversification tools present a viable solution for enhancing the precision of WebAssembly malware detection systems.

Defensive Software Diversification: Our techniques enhance overall security by facilitating the deployment of unique and diversified WebAssembly binaries,

[1] `https://github.com/bytecodealliance/wasm-tools`

potentially utilizing different variants as needed. For instance, WASM-MUTATE generates Wasm binaries that are resistant to Spectre-like attacks. Given that WASM-MUTATE can generate tens of thousands of variants within minutes, it becomes feasible to deploy a unique variant for each function invocation on FaaS platforms. This rationale applies equally to both CROW and MEWE. Our tools can mitigate other side-channel attacks. For example, CROW also excels at hardening defenses against potential side-channel attacks. In addition, MEWE tackles high-level timing-based side-channels [**?**].

5.3 Future Work

Along with this dissertation we have highlighted several open challenges related to Software Diversification in WebAssembly. These challenges open up several directions for future research. In the following, we outline some of these directions.

Extending WASM-MUTATE: WASM-MUTATE may gain advantages from the enumerative synthesis techniques employed by CROW and MEWE. Specifically, WASM-MUTATE could adopt the transformations generated by these tools as rewriting rules. This approach could enhance WASM-MUTATE in two specific ways. First, it could improve the preservation of the variants generated by WASM-MUTATE. Second, this method would inevitably expand the diversification space of WASM-MUTATE e-graphs.

Program Normalization: We successfully employed WASM-MUTATE for the evasion of malware detection (see Section 4.1). The proposed mitigation in the prior study involved code normalization as a means of reducing the spectrum of malware variants. Our current work provides insights into the potential effectiveness of this approach. Specifically, a practically costless process of pre-compiling Wasm binaries could be employed as a preparatory measure for malware classifiers. In other words, a Wasm binary can first be JITed to machine code, effectively eliminating malware variants. This approach could substantially enhance the efficiency and precision of malware detection systems.

Meta-oracles: Our experiment results in Section 4.1 indicate that VirusTotal surpasses MINOS in detecting WebAssembly cryptojacking. The primary factor contributing to this is VirusTotal's utilization of a broader range of antivirus vendors, which employs various detection strategies. On the other hand, MINOS functions as a binary oracle. This evidence supports the use of multiple malware oracles (meta-oracles) in identifying cryptojacking malware in browsers. In the context of WebAssembly, given the existence of numerous and diverse Wasm-specific detection mechanisms, this strategy is both practical and feasible, yet not explored in the literature.

Mitigating Port Contention: Rokicki et al. [**?**] showcased the potential of a covert side-channel attack using port contention in WebAssembly code

within browsers to violate cross scripting isolation. Side-channels exploiting port contention utilize the competition for shared hardware resources to extract sensitive information from processes. The attacker measures the time taken to access these shared resources to deduce data or behavior of a victim process sharing the same resources. Counteracting these attacks is especially difficult as they leverage fundamental features of the hardware design meant to enhance performance. The success of this attack largely relies on the accurate prediction of Wasm instructions inducing port contention. To tackle this security concern, WebAssembly Software Diversification can be effectively implemented as a browser plugin. Our tools possess the ability to change the WebAssembly instructions acting as port contention predictors with different instructions. This bears a strong resemblance to the impact on timers and padding discussed earlier in Section 4.2. Such a strategy would certainly eradicate the port contention in the particular port utilized for the attack, subsequently hardening browsers against such detrimental activities.

AI and Software Diversification: As discussed in Chapter 3, implementing a diversifier at the high language level seems impractical due to the multitude of existing frontends. However, the emergence of Large Language Models (LLMs) and their ability to generate high-level language may address this problem. Nevertheless, we argue that simply connecting the LLM to the diversifier does not provide a complete solution; studies on preservation must also be conducted. Specifically, high-level diversification might lead to low preservation, thereby challenging the assumption of diversification at the low-level. In the context of WebAssembly, considering the wide variety of frontends, utilizing LLMs might be a feasible method for generating Software Diversification. Although preservation poses a problem at a high-level, it could potentially solve the inherited, more challenging issue of transforming programs at the intermediate representation level or WebAssembly bytecode itself.